by
Andrew Schulman

Lab Notes

The Programming Challenge of Windows Protected Mode

■ Learning to use DPMI and a number of undocumented *Windows* services is the key to accessing real-mode memory from protected-mode *Windows* programs.

In last issue's Lab Notes, I explored and explained some of the benefits provided by *Windows*' use of the protected mode of the Intel 80286, 80386, and 80486 microprocessors (''Windows 3.0: All That Memory, All Those Modes''). Protected mode provides access to megabytes of memory, multitasking, and improved support for DOS applications. It also greatly simplifies memory management for *Windows* programmers. Indeed, from a programming viewpoint, *Windows* 3.0 is a protected-mode DOS extender.

Given Microsoft's market position and the rapid acceptance of *Windows* 3.0, protected-mode DOS must be considered the dominant operating environment for the next few years.

However, protected mode also has its nside. The Intel protection mechanism prevents program bugs from corrupting the system, but it also means that your programs can't just peek, poke, or call any old memory addresses they want. Accessing code and data outside your program becomes ''impossible.''

But if the vast majority of *Windows* programs are henceforth to be designed to run in protected mode, how then will they talk to device drivers, TSRs, and other precious resources that reside down in the first megabyte of memory, in real mode? In a nutshell, we must somehow accomplish the ''impossible.''

This issue's Lab Notes will examine the programming challenges software developers face as they try to move DOS programs to protected-mode *Windows*. You'll see why the keys to the solution lie in the new DOS Protected Mode Interface (DPMI) and in a number of undocumented features of the *Windows* application program interface (API).

In *PC Magazine*'s February 12, 26, and March 12, 1991, Power Programming columns, Ray Duncan showed how non-*Windows* DOS applications running under *Windows* can use DPMI. It is important to note that for these DOS programs, DPMI is (at least at present) available only in 386 Enhanced mode. For *Windows* programs, however, DPMI is available in both Standard and Enhanced modes. In this article we'll see how and when a *Windows* application can access DPMI.

THE PROBLEM WITH PROTECTION

Because it offers programs a potentially enormous address space and because it supports multitasking, protected mode must establish some ''rules of the road'' that were not necessary in the little 640K sandbox of real-mode DOS. These rules are the software equivalents of ''keep your hands to yourself,'' ''don't snoop,'' and ''don't chew with your mouth open.'' That is, your programs can't peek (read), poke (write), or call (execute) memory that isn't mapped into their own address space. Furthermore, they can't execute data and they can't overwrite code. Protected-mode programs can allocate megabytes of memory, but they can't access memory that isn't theirs.

But many programs on the PC *need* low-level access to code and data that isn't theirs. Key system variables, for example, are kept in the BIOS data segment at memory location 400h (which can be represented by many real-mode addresses, including 0000:0400 and 0040:0000). Needless to say, this address is not located inside your program.

You can, of course, *try* to read this area from your protected-mode *Windows* program, using something like the following C code:

```
unsigned long far *pticks =
  (unsigned long far *) 0x46C;
unsigned long ticks = *pticks;
```

This code works in real mode, but in protected mode the variable `ticks` does *not* end up receiving the BIOS timer-tick count. What you get instead is the following message from *Windows*:

```
UNRECOVERABLE APPLICATION ERROR
Terminating current application.
```

Not so good.

If you run this short program within a debugger (CodeView for *Windows* (CVW) or Turbo Debugger for *Windows* (TDW)), your application will not be terminated. Instead, the debugger will display a message such as ''Trap 13 (0DH) - General Protection Fault.'' If there were some way to repair the problem while still in the debugger, you'd be able to resume the program. Usually that isn't possible, so there's no way to continue past the line of code that violated protection. But at least the debugger shows you where the problem is.

For a software developer, CVW's ''General Protection Fault'' is a much more informative message than *Windows*' ''Unrecoverable Application Error''

(UAE). A general protection fault (GP fault) is generated by the Intel processor whenever the rules of protected mode have been violated. An operating environment installs a handler for this INT 0Dh exception. In *Windows*' case, the handler responds by shutting down the offending application and displaying the rather uninformative UAE message box.

Up to this point, we have seen that whenever a DOS program that has been moved to protected mode tries to access an external real-mode memory location such as 0040:0000, it generates a GP fault. So when you move to protected-mode *Windows*, how *do* you continue to access the resources you used in real mode? Besides the BIOS data area, for example, how would you communicate with a database package, TSR, or a network driver? How could you access the undocumented DOS data structures that *PC Magazine* utilities frequently use? Is there no way to take these with you when you cross the border into protected mode?

This is vitally important, for it will be many years before every driver, TSR, and memory-mapped device is made directly accessible in protected mode. Until then, developers *must* be able to access real-mode services and absolute memory locations from their protected-mode programs. Fortunately, as we'll see, the low-level aspects of the machine *are* still visible from protected mode. And programmers who can write protected-mode code that talks to real mode will find that they have an increasingly marketable skill.

REAL-MODE DEV CODE — **COMPLETE LISTIN**

AVAILABLE ON PC MAGNET

```
ListOfLists far *doslist;
DeviceDriver far *dd;
// ...
dd = &doslist->nul;
for (;;)
{
    printf("%Fp\t", dd);
    if (dd->attr & CHAR_DEV)
        printf("%.8Fs\n", dd->u.name);
    else
        printf("Block dev: %u unit(s)\n", dd->u.blk_cnt);
    dd = dd->next;
    if (FP_OFF(dd->next) == -1)
        break;
}
```

Figure 2: The C code for walking the DOS device list in real mode.

WALKING THE DEVICE CHAIN

Consider a common DOS program that simply displays the names and addresses of all device drivers loaded on your system. Both Turbo C++ and Borland C++ come with such a program—called TDDEV—and Quarterdeck Office Systems' *Manifest* includes a similar display.

A DOS program like TDDEV can, of course, be run under *Windows*. But how would you turn this into an actual *Windows* program, so that it can interact with the user instead of simply dumping information out to the screen? Eventually, you might want to make it possible for the user to click on a device name to obtain more-detailed information. All sorts of possibilities open up once it's a genuine *Windows* program. But for now let's be content with the same, noninteractive output, changing the program only so that it runs on the *Windows* side of the fence rather than in a DOS box. The output of such a *Windows* DEV program is shown in Figure 1.

One issue to be faced in contemplating the necessary changes is how to structure the *Windows* source code so that it continues to resemble, as closely as possible, our old character-based DOS code. For example, it would be nice to continue using the C printf() function and to avoid (at least for the time being) the complexities of *Windows* event-driven programming. (This issue is later explored in the sidebar "Windows and printf().")

In *Windows* 3.0, the program will need to run in the Standard and 386 Enhanced modes of *Windows*. That, in turn, me running in the protected mode of the Intel 80286, 80386, or 80486. A DEV program must walk the DOS "device chain," printing out the name and address of each driver found on this linked list. The challenge in porting such a program to protected-mode *Windows* is that the DOS device chain is a real-mode data structure, located in conventional memory. We must somehow access this data structure from protected-mode *Windows*. We have already seen that attempting such access normally produces a UAE or GP fault.

It's easy to produce this information in a real-mode DOS program. First, you make the undocumented DOS call INT 21h function 52h to get a segment:offset far pointer to the DOS internal variables table (sometimes called the List of Lists). The header for the NUL device is stored right inside this table, and the NUL device is the anchor for the entire DOS device chain. It is called a *chain* because each device header contains the address of the *next* device; it is a linked list.

In Figure 1, NUL is at the real-mode address 0000:1228. It contains a pointer t the next device, which in this example *SmartDrive* (SMARTAAR). The pointer is simply a 4-byte number, 0000:8B80, the address for SMARTAAR. SMART-

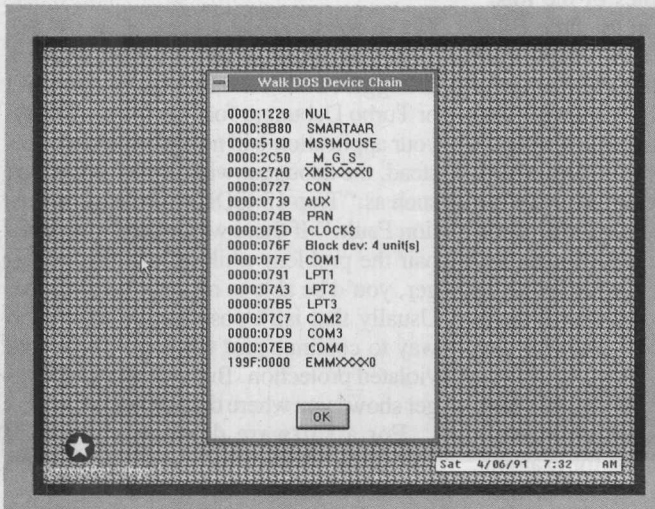

Figure 1: DEV.EXE, a protected-mode *Windows* program, produces an output similar to a normal real-mode DOS program.

AAR's device header in turn contains a pointer, 0D34:0000, to the Microsoft Mouse (MS$MOUSE). The list continues in this fashion until it gets to a Lotus/Intel/-Microsoft Expanded Memory Manager (EMMXXXX0), whose device header lives at real-mode address 199F:0000 and which, incidentally, has been patched into the DOS device chain by *Windows* in Enhanced mode. Here the "next" field contains a -1 (FFFFh) in its offset portion. You know you've reached the end of the list when you find a "next" field whose offset is FFFF.

As the program walks this list, it prints out the driver's address, and either the name (for character devices) or the number of units the device provides (for block devices). For both cases, this information is found in the device header. In a C program written for real-mode DOS, a loop that walks the device chain would look something like the code shown in Figure 2.

There is a direct correspondence between this code and the output shown in ure 1. But that display comes from a protected-mode *Windows* program. How did the program access these real-mode addresses from protected mode? Why didn't it terminate with a UAE or a GP fault?

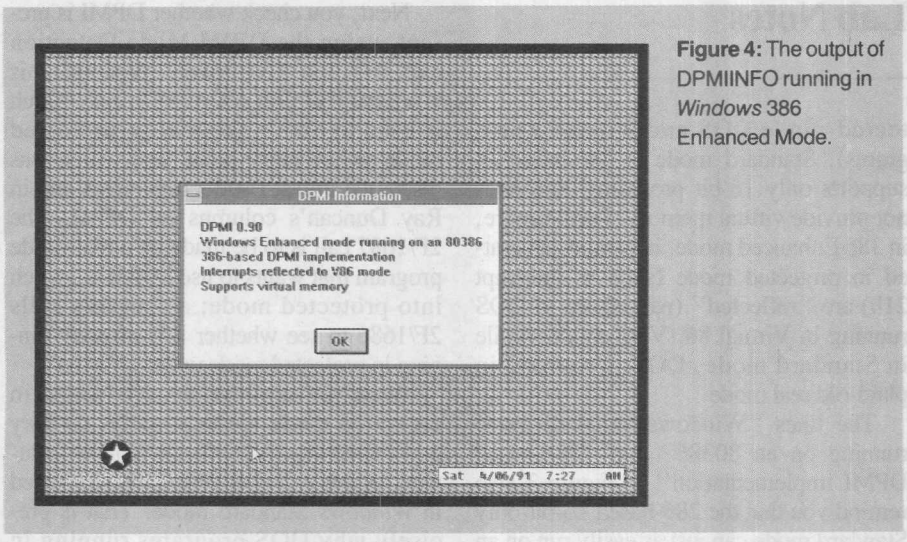

Figure 4: The output of DPMIINFO running in *Windows* 386 Enhanced Mode.

ENTER DPMI

As I mentioned earlier, by utilizing the DOS Protected Mode Interface (DPMI) and some undocumented *Windows* services, protected-mode *Windows* programs *can* do all the low-level things they need to do, without breaking the rules of protected mode. In protected mode, DPMI uses interrupt 31h to provide a set of services for calling real-mode code, mapping real-mode memory locations into the protected-mode address space, hooking real-mode interrupts, and the like.

Unlike the EMS and XMS specifications, DPMI was never designed for widespread use. It was intended for system software, not application programs. The DPMI specification was drawn up by a committee that included Microsoft, Intel, Lotus, Phar Lap, Quarterdeck, Rational Systems, IBM, Borland, and other companies, and the assumption was that this handful of companies would constitute DPMI's user base.

The true purpose of DPMI is to allow the peaceful coexistence of protected-mode DOS extenders, expanded memory managers, and DOS-based multitasking environments. As with the earlier VCPI (Virtual Control Program Interface) specification developed by Quarterdeck and Phar Lap, users—and the vast majority of programmers—are expected to care only whether the products they use support the specification. As for the details of the specification itself, they couldn't care less.

That was the theory. What happened, however, is that *Windows* programmers discovered a major gap in the *Windows* application programmer's interface (API). Among all the 500-plus functions listed in the massive *Windows Programmer's Reference* there is nothing that lets you call a DOS device driver, for example. You can't even find its location in memory.

DPMI, on the other hand, *does* provide services that allow low-level access to the machine from protected mode. And DPMI is incorporated in *Windows*. Consequently, many programmers look to DPMI (and to a number of useful, but undocumented *Windows* functions and features) as the back door to *Windows* 3.0 (to coin a mixed metaphor).

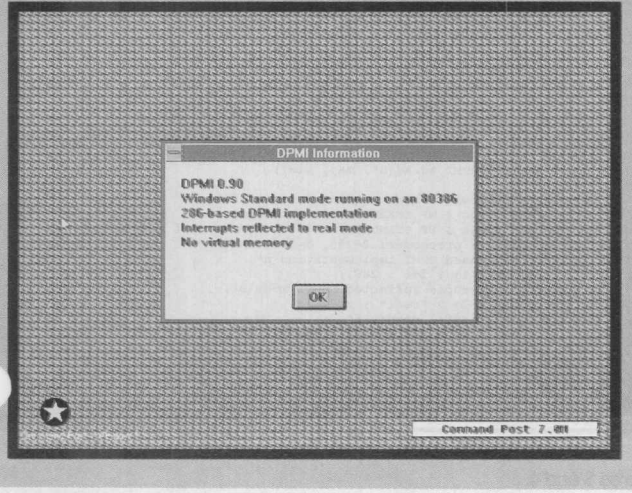

Figure 3: The output of DPMIINFO running in *Windows* Standard Mode.

CALLING DPMI

Before jumping in and applying DPMI directly to our DEV program, let's sample a more easily digested example of DPMI in a *Windows* program. DPMIINFO is a small utility that simply displays a few interesting facts about whatever DPMI implementation is present. The screenshots of DPMIINFO shown in Figures 3 and 4 summarize some of the crucial differences between *Windows* Standard and 386 Enhanced modes. While these were discussed in last issue's Lab Notes, they're worth reviewing here.

Whereas 386 Enhanced mode supports virtual memory (paging to disk) and is 386-based (that is, it supports 32-bit pro-

tected-mode DOS and *Windows* programs), Standard mode is 286-based (it supports only 16-bit programs) and does not provide virtual memory. Furthermore, in 386 Enhanced mode, interrupts generated in protected mode (such as interrupt 21h) are "reflected" (passed on) to DOS running in Virtual 86 (V86) mode, while in Standard mode, DOS is running in plain-old real mode.

The lines "Windows Standard mode running on an 80386" and "286-based DPMI implementation" in Figure 3 also remind you that the 286-based 16-bit-only Standard mode can just as easily run on an 80386 processor as on an actual 80286.

Finally, Figures 3 and 4 also indicate that *Windows* 3.0 uses DPMI, Version 0.9. While the specifications for DPMI 1.0 have been drawn up, it's DPMI 0.9 that most end users will have on their machines for some time to come. (The DPMI 0.9 specification is available free of charge from Intel. Call 1-800-548-4725 and ask for Part No. 240763.)

Turning to the DPMIINFO.C source code, which is shown in Figure 5, we see that there are several tests that should be made before you call the DPMI INT 31h functions.

First, you should see whether you are running in protected mode. We use the *Windows* API GetWinFlags() function for this.

Next, you check whether DPMI is present, using the DPMI Mode Detection call, INT 2Fh AX=1686h. (Note that this is *not* the INT 2Fh AX=1687h call, which is used to obtain the real-to-protected mode switch entry point, and which was discussed in last issue's Lab Notes and in Ray Duncan's columns on DPMI. The 2F/1687 call is to be made by a real-mode program that wants to use DPMI to switch into protected mode; a program calls 2F/1686 to see whether it is *already* running in protected mode under DPMI.)

Windows applications are loaded in protected mode automatically, so they don't need the 2F/1686 call. This is fortunate, as this call is not currently supported in *Windows* Standard mode. That is precisely why DOS programs running in Standard mode can't access DPMI but *Windows* programs can.

Once you have determined that you are running in protected mode under DPMI, you can make interrupt 31h calls: INT 31h services are provided in protected mode only. In Figure 5, then, we call the DPMI Get Version function (INT 31h AX=0400h). The functions dpmi_present() and dpmi_version() called by DPMIINFO.C in Figure 5 are both provided (along with many other DPMI-related functions) by DPMI.H and DPMI.C, whose code listings are shown in Figures 6 and 7 respectively.

Note that DPMI.C (in Figure 7) uses in-line assembly language to make INT 2Fh and INT 31h calls. Microsoft C 6.0 and Borland C++ 2.0 both provide an in-line assembler, and DPMI.C will work with either compiler. We avoid the C int8(and int86x() functions here, because many implementations of these functions cause GP faults in protected mode—which is precisely what we're trying to avoid by going to DPMI!

In addition to the functions in DPMI.C and DPMI.H, you may want to look into one of several DPMI libraries now being sold. For example, SoftWorks International (404-876-6115) sells a library called *SoftDPMI*.

CALLING REAL-MODE SERVICES

We are now in a position to see how DPMI helps move the DEV program to *Windows*. In DEV.C (Figure 8), several blocks of code have been marked

```
#ifdef WINDOWS
```

This code is used when you build a program with -DWINDOWS on the compiler command line.

The first such block appears in the function get_doslist(). This is the function that, as noted earlier, calls the undocumented DOS function INT 21h AH=52h and returns a far pointer to the DOS internal va ables table.

When it is being compiled for *Windows*, get_doslist() calls the function dpmi_rmode_intr(), which is declared in DPMI.H (Figure 6) and implemented in DPMI.C (Figure 7). The dpmi_rmode_intr() function simply provides a C interface to the DPMI Simulate Real Mode In-

DPMIINFO.C — **COMPLETE LISTING**

```
/*
DPMIINFO.C -- display DPMI information under Windows

Copyright (c) 1991 Ziff Communications Co.
    PC Magazine * Andrew Schulman

Borland C++ 2.0:
    bcc -W -2 dpmiinfo.c printf.c dpmi.c
    rc dpmiinfo.exe

Microsoft C 6.0:
    cl -c -AS -G2sw -Oais -Zpe dpmiinfo.c printf.c dpmi.c
    link /align:16 dpmiinfo printf dpmi,dpmiinfo,,/nod slibcew libw,wi|n.def
    rc dpmiinfo.exe
*/

#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <dos.h>
#include <windows.h>
#include "printf.h"
#include "dpmi.h"

#define FAIL(s) MessageBox(NULL, s, "DPMI Information", MB_OK)

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpszCmdLine, int ncmdShow)
{
    unsigned long win_flags;
    unsigned flags;
    unsigned maj, min;
    unsigned proc;

    if (! ((win_flags = GetWinFlags()) & WF_PMODE))
        return FAIL("This program requires Windows "
                    "Standard or Enhanced mode");

    if (! dpmi_present())
        return FAIL("DPMI not present");

    dpmi_version(&maj, &min, &flags, &proc);

    open_display("DPMI Information");

    if (win_flags & WF_STANDARD)
        printf("DPMI %x.%x\n", maj, min); // silly bug in Standard mode
    else
        printf("DPMI %d.%d\n", maj, min);

    printf("Windows %s mode running on an 80%d86\n",
        (win_flags & WF_ENHANCED) ? "Enhanced" :
        (win_flags & WF_STANDARD) ? "Standard" : "Not Another?!",
        proc); // processor: 2=286, 3=386, 4=486
    printf("%d-based DPMI implementation\n",
        (flags & 1) ? 386 : 286);
    printf("Interrupts reflected to %s mode\n",
        (flags & 2) ? "real" : "V86");
    printf("%s virtual memory\n",
        (flags & 4) ? "Supports" : "No");

    show_display();
    return 0;
}
```


Figure 5: The DPMIINFO program displays the DPMI information shown in Figures 3 and 4.

DPMI.H — **COMPLETE LISTING**


```
/* DPMI.H */

#pragma pack(1)

typedef struct {
    unsigned long edi, esi, ebp, reserved, ebx, edx, ecx, eax;
    unsigned flags, es, ds, fs, gs, ip, cs, sp, ss;
    } RMODE_CALL;

/* structure of a protected-mode descriptor */
typedef struct {
    unsigned limit, addr_lo;
    unsigned char addr_hi, access, reserved, addr_xhi;
    } DESCRIPTOR;

/* low-level DPMI functions */
BOOL dpmi_present(void);
void dpmi_version(unsigned *pmaj, unsigned *pmin,
    unsigned *pflags, unsigned *pproc);
BOOL dpmi_rmode_intr(unsigned intno, unsigned flags,
    unsigned copywords, RMODE_CALL far *rmode_call);
unsigned dpmi_sel(void);
BOOL dpmi_set_descriptor(unsigned pmodesel, DESCRIPTOR far *d);
BOOL dpmi_get_descriptor(unsigned pmodesel, DESCRIPTOR far *d);
BOOL dpmi_sel_free(unsigned pmodesel);

/* higher layer on top of DPMI */
unsigned DosAllocRealSeg(DWORD bytes, unsigned *ppara, unsigned *psel);
unsigned DosFreeRealSeg(unsigned sel);
unsigned DosMapRealSeg(unsigned rmpara, DWORD size, unsigned far *psel);
unsigned DosFreeSeg(unsigned sel);
void far *DosProtToReal(void far *prot);

void far *map_real(void far *rptr, unsigned long size);
void free_mapped_seg(void far *fp);
unsigned get_mapped(void);
unsigned verw(unsigned sel);

/* undocumented Windows functions */
extern DWORD FAR PASCAL GetSelectorBase(unsigned sel);
extern DWORD FAR PASCAL GetSelectorLimit(unsigned sel);
extern void FAR PASCAL SetSelectorBase(unsigned sel, DWORD base);
extern void FAR PASCAL SetSelectorLimit(unsigned sel, DWORD limit);
```

Figure 6: This is the DPMI header file.

terrupt function, INT 31h AX=0300h—one of the most important "translation services" provided by DPMI.

Wait a minute, though. Simulate Real Mode Interrupt? Translation service? Why can't we call INT 21h AH=52h via the C intdosx() function?

The problem is this: The *Windows* version of DEV.C is running in protected mode. INT 21h AH=52h is an MS-DOS function that runs in real mode. Since *Windows* is a DOS extender, one of its primary responsibilities is to provide INT 21h functions in protected mode so that *Windows* apps can open files, allocate memory, change subdirectories, and so on, simply by making INT 21h calls. But INT 21h AH=52h is a special case. Since it's an undocumented function, you have no idea whether or not *Windows* provides direct access to it in protected mode.

That's why you must invoke this function via the DPMI Simulate Real Mode Interrupt function. As can be seen in the get_doslist() function in Figure 8, a program loads up an image of the 32-bit CPU registers in an RMODE_CALL structure and then calls dpmi_rmode_intr(). Using dpmi_rmode_intr() is much like using the int86() function in Microsoft C and other compilers.

The point of this exercise is not simply to show how you make undocumented DOS calls from a *Windows* application. For all we know, INT 21h AH=52h may *be* supported in protected-mode *Windows*. Furthermore, a DOS extender can provide direct access to the undocumented DOS functions from protected mode. For example, Version 3.0 of Phar Lap's 386 DOS-Extender provides this and many other undocumented DOS functions in protected mode, so it eliminates the need for a Simulate Real Mode Interrupt function.

However, there will always be *some* functions to which a DOS extender does not provide direct access. The INT 21h AH=52h example is representative of that larger problem. If there is an interrupt-based service that you need to call and it is not supported directly in protected-mode *Windows*, then you must use INT 31h AX=0300h, probably via a C interface like dpmi_rmode_intr() as indicated above. DPMI also provides similar functions for calling stack-based real-mode procedures.

ACCESSING REAL-MODE MEMORY

The get_doslist() function returns a far pointer to the DOS ListOfLists structure. But *what kind* of far pointer? When writing protected-mode programs for a fundamentally real-mode operating system like DOS, you must accustom yourself to asking, "Is this a real-mode pointer or a protected-mode pointer?" In the case of get_doslist(), INT 21h AH=52h returns a real-mode pointer in ES:BX even if you invoke it via the dpmi_rmode_intr() function.

You now have a real-mode pointer, and you want to examine the corresponding block of memory in protected mode. Thus, the next important #ifdef WINDOWS block in DEV.C calls the map_real() function. This function's job is to take a block of memory (whose size and real-mode address are passed as parameters) and return an equivalent protected-mode pointer.

Any time you have a *Windows* program that needs to peek or poke a piece of real-mode memory, you can use the map_real() function in DPMI.H and DPMI.C (along with its complement, the free_mapped_seg() function) without knowing anything about the underlying complexities of DPMI. Here, however, you may be interested in how map_real() works.

INSIDE MAP_REAL()

In DPMI.C, map_real() is built on top of the DosMapRealSeg() function, which in turn is built on top of several low-level C interfaces to DPMI functions.

The reason for all these layers is that the DPMI specification is extremely low level. Recall that the intent of DPMI was to provide services for a handful of DOS extender vendors to assure compatibility among their protected-mode extensions to MS-DOS. DPMI's developers had no idea that large numbers of *Windows* programmers would need to call DPMI services directly. Thus, it is best to construct several higher-level layers on top of DPMI.

The actual DPMI functions (with their

```
/*
DPMI.C -- DPMI functions, plus some undocumented Windows functions that
provide the same functionality, plus several layers of "sugar coating"
on top of these low-level services, to make them palatable

Copyright (c) 1991 Ziff Communications Co.
    PC Magazine * Andrew Schulman
*/

#include <windows.h>
#include <dos.h>
#include "dpmi.h"

#define MAKEP(seg, ofs) ((void far *) MAKELONG((ofs), (seg)))

BOOL dpmi_present(void)
{
    _asm mov ax, 1686h
    _asm int 2fh
    _asm not ax
}

void dpmi_version(unsigned *pmaj, unsigned *pmin,
    unsigned *pflags, unsigned *pproc)
{
    unsigned char maj, min, proc;
    unsigned flags;
    _asm {
        mov ax, 0400h
        int 31h
        mov maj, ah
        mov min, al
        mov flags, bx
        mov proc, cl
        }
    *pmaj = maj;
    *pmin = min;
    *pflags = flags;
    *pproc = proc;
}

/* Performs a real-mode interrupt from protected mode */
BOOL dpmi_rmode_intr(unsigned intno, unsigned flags,
    unsigned copywords, RMODE_CALL far *rmode_call)
{
    if (flags) intno |= 0x100;
    _asm {
        push di
        mov ax, 0300h       // simulate real-mode interrupt
        mov bx, word ptr intno      // interrupt number, flags
        mov cx, word ptr copywords  // words to copy from pmode to rmode stack
        les di, dword ptr rmode_call  // ES:DI = addr of rmode call struct
        int 31h             // call DPMI
        jc error
        mov ax, 1           // return TRUE
        jmp short done
        }
error:
        _asm xor ax, ax     // return FALSE
done:
        _asm pop di
}

/* Allocates a single protected-mode LDT selector */
unsigned dpmi_sel(void)
{
    _asm {
        xor ax, ax          // Allocate LDT Descriptors
        mov cx, 1           // allocate just one
        int 31h             // call DPMI
        jc error
        jmp short done      // AX holds new LDT selector
        }
error:
        _asm xor ax, ax     // failed
done:;
}

BOOL dpmi_set_descriptor(unsigned pmodesel, DESCRIPTOR far *d)
{
    _asm {
        push di
        mov ax, 000ch       // Set Descriptor
        mov bx, word ptr pmodesel    // protected mode selector
        les di, dword ptr d // descriptor
        int 31h             // call DPMI
        jc error
        mov ax, 1           // return TRUE
        jmp short done
        }
error:
        _asm xor ax, ax     // return FALSE
done:
        _asm pop di
}

BOOL dpmi_get_descriptor(unsigned pmodesel, DESCRIPTOR far *d)
{
    _asm {
        push di
        mov ax, 000bh       // Get Descriptor
        mov bx, word ptr pmodesel    // protected mode selector
        les di, dword ptr d // descriptor
        int 31h             // call DPMI
        jc error
        mov ax, 1           // return TRUE
        jmp short done
        }
error:
        _asm xor ax, ax     // return FALSE
done:
        _asm pop di
}

BOOL dpmi_sel_free(unsigned pmodesel)
{
    _asm {
        mov ax, 0001h       // Free LDT Descriptor
        mov bx, word ptr pmodesel    // selector to free
        int 31h             // call DPMI
        jc error
        mov ax, 1           // return TRUE
        jmp short done
        }
error:
        _asm xor ax, ax     // return FALSE
done:;
}

/* Layer on top of DPMI and Windows services *****************************/

unsigned DosAllocRealSeg(DWORD bytes, unsigned *ppara, unsigned *psel)
{
    DWORD dw = GlobalDosAlloc(bytes);
    if (dw == NULL)
        return 8;   /* insufficient memory */
    *ppara = HIWORD(dw);
    *psel = LOWORD(dw);
    return 0;
}

unsigned DosFreeRealSeg(unsigned sel)
{
    return (GlobalDosFree(sel) != NULL);
}

/*
    Use DPMI services to map a real-mode paragraph into protected-mode
    address space. First we get a selector using the DPMI "Allocate
    LDT Descriptors" call (INT 31h Function 0000h), via our dpmi_sel()
    function. We then get the descriptor for any old data segment in
    our program so that it can be used as a template of sorts for the
    new descriptor. This is done with the DPMI "Get Descriptor" call
    (INT 31h Function 000Bh), via our dpmi_get_descriptor() function.
    We then alter the descriptor to reflect the "rmpara" and "size"
    requested, and finally associate the descriptor with our LDT
    selector, using the DPMI "Set Descriptor" call (INT 31h Function
    000Ch). All the user needs, of course, is the selector itself.
*/
unsigned DosMapRealSeg(unsigned rmpara, DWORD size, unsigned far *psel)
{
    DESCRIPTOR d;
    unsigned long addr;
    unsigned sel = dpmi_sel();
    if (! sel)
        return 8;   /* insufficient memory */
    /* make sure psel is valid */
    if (! verw(FP_SEG(psel)))
        return 490; /* invalid selector error */
    /* get descriptor for any data segment */
    dpmi_get_descriptor(FP_SEG(psel), &d);
    d.limit = (unsigned) size - 1;
    addr = ((unsigned long) rmpara) << 4L;
    d.addr_lo = (unsigned) addr;
    d.addr_hi = (unsigned char) (addr >> 16);
    d.reserved = d.addr_xhi = 0;
    dpmi_set_descriptor(sel, &d);
    *psel = sel;
    return 0;       /* success */
}
unsigned DosFreeSeg(unsigned sel)
{
    return ! dpmi_sel_free(sel);
}

/*
    Use undocumented Windows function to retrieve the real-mode
    equivalent to a protected mode pointer. Of course, we could also
    have used dpmi_get_descriptor() here, but GetSelectorBase()
    is slightly more convenient. If the base of the selector is
    above the one-megabyte watershed, then there is no real-mode
    equivalent, so we return NULL.
*/
void far *DosProtToReal(void far *prot)
{
    unsigned long base = GetSelectorBase(FP_SEG(prot));
    if (base > 0xFFFFFL)
        return NULL;    /* not accessible in real mode */
    else
        return MAKEP(base >> 4, (base & 0x0F) + FP_OFF(prot));
}

unsigned __0000H = 0;   /* undocumented Windows selector */
```

Figure 7: The source code for DPMI.EXE.

DPMI.C 2 of 2

```
unsigned mapped = 0;    /* keep track of number of mapped selectors */

unsigned get_mapped(void)   { return mapped; }

/*
    Map a real-mode pointer into our protected mode address space.
    If the real-mode pointer is in the first 64k of memory, use the
    undocumented Windows selector __0000H. Otherwise, use DPMI
    services via our DosMapRealSeg() function, which provides a more
    convenient layer on top of DPMI. This map_real() function in
    turn provides a more convenient layer on top of DosMapRealSeg().
    Note that DosMapRealSeg() takes a paragraph address, whereas
    map_real takes a full segment:offset real-mode pointer.
*/
void far *map_real(void far *rptr, unsigned long size)
{
    unsigned seg, ofs, sel;

    if (! __0000H)  /* one time init: get undocumented Windows selector */
        __0000H = LOWORD(GetProcAddress(GetModuleHandle("KERNEL"),"__0000H"));

    seg = FP_SEG(rptr);
    ofs = FP_OFF(rptr);
    if ((seg < 0x1000) && ((ofs + size) < 0xFFFF))
        return MAKEP(__0000H, (seg << 4) + ofs);
    if (DosMapRealSeg(seg, size + ofs, &sel) != 0)
        return 0;
    mapped++;
    return MAKEP(sel, ofs);
}

void free_mapped_seg(void far *fp)
{
    unsigned sel = FP_SEG(fp);
    if (sel == __0000H)
        return;
    if (DosFreeSeg(sel) == 0)
        mapped--;
}

/* Use Intel VERW instruction to validate pointers */
unsigned verw(unsigned sel)
{
    _asm mov ax, 1
    _asm verw sel
    _asm je short ok
    _asm xor ax, ax
ok:;
}
```

C interfaces in DPMI.C) used when DEV.C calls map_real() are these:

- Allocate LDT Descriptors (INT 31h AX=0000h); dpmi_sel()
- Get Descriptor (INT 31h AX=000Bh); dpmi_get_descriptor()
- Set Descriptor (INT 31h AX=000Ch); dpmi_set_descriptor()

LDT? Descriptor? It's hard to see what this has to do with accessing real-mode absolute memory locations from a protected-mode program. In last issue's Lab Notes, however, you'll recall that in protected mode, the *XXXX* portion of an *XXXX*:*YYYY* pointer is called a selector, and that this selector is effectively an index into what is called a "descriptor table." This is in contrast to real mode, in which an *XXXX*:*YYYY* pointer (1234:0005, for example) is merely a slightly warped representation of an absolute memory address like 12345h.

Memory addressing in protected mode is indirect. Thus, the absolute memory address that corresponds to a protected-mode pointer of 1234:0005 depends entirely on the "base address" that is stored in the descriptor corresponding to selector 1234h. It bears no relation to the value of the selector itself. In *Windows* 3.0, all *Windows* programs share a common table of such descriptors, called the Local Descriptor Table (LDT). All memory references from *Windows* programs essentially go through the LDT.

This indirection is part of the mechanism that the Intel processors use to enforce protection. It is also why we can't peek at addresses such as 12345h simply by forming a pointer that looks like 1234:0005 (or, more likely, peek at an address such as 046Ch simply by forming a pointer that looks like 0040:006C or 0000:046C). In protected-mode *Windows*, selectors such as 1234h and 0040h are simply being used as indexes for table lookup in the LDT. (Technically, this is something of an oversimplification of the wonderfully baroque mechanism actually used in the Intel processors. For the actual bit-by-bit details, see Ray Duncan et al., *Extending DOS* (Reading Mass.: Addison-Wesley, 1990, or Jeff Prosise, "Segmented Memory," *PC Magazine*, March 26, 1991.))

In *Windows* 3.0, all *Windows* programs share a table of descriptors known as the Local Descriptor Table, and all memory references essentially pass through it.

Fortunately, however, the indirection does more than cause a problem for protected-mode *Windows* applications; it also provides the solution to the problem. After all, if you want to examine address 46Ch, for example, you don't really care what the *XXXX*:*YYYY* pointer looks like. If there were a protected-mode descriptor somewhere whose base address was 400h, you could use a selector that corresponded to this descriptor, even if the selector value itself bore no resemblance to the number 40, 400, or whatever. And you could always save the selector in a variable called my40, bios_seg, or perhaps __0040H.

In order to access absolute memory locations in protected mode, then, what you need is to get a selector whose associated descriptor has a base address, limit, and access-rights that correspond to the absolute address you're interested in. And that's exactly the sort of service DPMI provides.

In the DPMI.C program, the DosMapRealSeg() function first calls dpmi_sel() to allocate an LDT descriptor and return its corresponding selector. Next it calls dpmi_get_descriptor() to get a descriptor for *any* data segment in your program, to serve as a kind of "template" for the new descriptor. Then it alters the descriptor to reflect the base address and size requested (for example, sizeof(DeviceHeader) bytes starting at 106E:0000). Finally, you install this altered descriptor using the dpmi_set_descriptor() call. Presto! The selector now corresponds to a descriptor with the base-address and size that the caller requested. The only thing that the caller needs back is the selector, which the function returns.

As I mentioned earlier, you can use the map_real() function without understanding any of this explanation, but knowledge of how protected mode works and of how to use DPMI is going to be increasingly valuable.

Lab Notes

A LIMITED RESOURCE

Returning to the code for DEV.C (Figure 8), note that the map_real() function is called in a loop as DEV walks the DOS device chain. This loop—which is actually the central part of the program—has been excerpted for greater clarity in Figure 9, and can be contrasted with the real-mode code shown previously in Figure 2. Comparing Figure 2 with Figure 9 will show you the major difference between the real-mode and protected-mode versions of the program.

In addition to calling map_real() in the loop, the free_mapped_seg() function in DPMI.C is also called upon to free up mapped selectors. This is extremely important, because descriptors and their corresponding selectors are a precious limited resource. Remember, all *Windows* programs (at least in Version 3.0) share a common LDT, and this can hold a maximum of only 8,192 descriptors.

Freeing mapped selectors is so important that the map_real() and free_mapped_seg() functions in DPMI.C keep track of how many outstanding mapped selectors there are. Just before DEV.C is ready to exit, it checks to make sure no mapped selectors have been left behind, for these are *not* automatically freed when a *Windows* program terminates.

UNDOCUMENTED WINDOWS

If you recall its original purpose, the DE program needs to print out not only the name of each DOS device driver, but also its address. Although DEV in *Windows uses* a protected-mode address returned from the map_real() function, this is not the address you want printed to the display. The protected-mode address is an arbitrary value that bears no relation at all to the real-mode address that would make sense to the DEV program user.

Thus, you want to display the real-mode address that corresponds to the protected-mode address. Since this real-mode address is whatever value was passed into map_real() in the first place, you could, of course, have saved it away somewhere for

DEV.C 1 of 2

```
/*
DEV.C -- display MS-DOS device chain

Copyright (c) 1991 Ziff Communications Co.
    PC Magazine * Andrew Schulman

real mode:
    Borland C++ 2.0: bcc dev.c
    Microsoft C 6.0: cl dev.c

Borland C++ 2.0 (DPMI.C *must* be compiled with -B flag):
    bcc -W -DWINDOWS -2 -B dev.c printf.c dpmi.c
    rc dev.exe

Microsoft C 6.0:
    cl -c -AS -G2sw -Oais -Zpe -W3 -DWINDOWS dev.c printf.c dpmi.c
    link /align:16 dev dpmi printf,dev,,/nod slibcew libw,win.def
    rc dev.exe

WIN.DEF:
    ; WIN.DEF -- generic Windows .DEF file
    EXETYPE         WINDOWS
    STUB            'WINSTUB.EXE'
    CODE            PRELOAD MOVEABLE DISCARDABLE
    DATA            PRELOAD MOVEABLE MULTIPLE
    HEAPSIZE        10240
    STACKSIZE       5120
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>
#ifdef WINDOWS
#include <windows.h>
#include "dpmi.h"
#include "printf.h"
#endif

#ifdef WINDOWS
char *app = "Walk DOS Device Chain";
#define puts(s)   MessageBox(NULL, s, app, MB_OK)
#endif
#define fail(s)   return puts(s)

#ifndef MK_FP
#define MK_FP(seg, ofs) \
    ((void far *) (((unsigned long) (seg) << 16) | (ofs)))
#endif

/* some device attribute bits */
#define CHAR_DEV    (1 << 15)
#define INT29       (1 << 4)
#define IS_CLOCK    (1 << 3)
#define IS_NUL      (1 << 2)

#pragma pack(1)

typedef struct DeviceDriver {
    struct DeviceDriver far *next;
    unsigned attr;
    unsigned strategy;
    unsigned intr;
    union {
        unsigned char name[8];
        unsigned char blk_cnt;
        } u;
    } DeviceDriver;

typedef struct {
    unsigned char misc[8];
    DeviceDriver far *clock;
    DeviceDriver far *con;
    unsigned char misc2[18];
    DeviceDriver nul;   /* not a pointer */
    // ...
    } ListOfLists;  // DOS 3.1+

ListOfLists far *get_doslist(void)
{
#ifdef WINDOWS
    /*
        Call undocumented DOS INT 21h Function 52h via DPMI "Simulate
        Real Mode Interrupt" call (INT 31h AX=0300h), and return
        the resulting real-mode pointer
    */
    RMODE_CALL r;
    memset(&r, 0, sizeof(r));
    r.eax = 0x5200;
    return (dpmi_rmode_intr(0x21, 0, 0, &r)) ? MK_FP(r.es, r.ebx) : 0;
#else
    union REGS r;
    struct SREGS s;
    segread(&s);
    s.es = r.x.bx = 0;
    r.h.ah = 0x52;
    intdosx(&r, &r, &s);
    return MK_FP(s.es, r.x.bx);
#endif
}

#ifdef WINDOWS
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpszCmdLine, int nCmdShow)
#else
int main(int argc, char *argv[])
#endif
{
    ListOfLists far *doslist;
    DeviceDriver far *dd;
#ifdef WINDOWS
    DeviceDriver far *next;
    int mapped;
#endif

#ifdef WINDOWS
    if (! (GetWinFlags() & WF_PMODE))
        fail("This program requires Windows Standard or Enhanced mode");

    if (! dpmi_present())
        fail("This program requires DPMI INT 31h services");
#endif

    if (! (doslist = get_doslist()))
        fail("INT 21h Function 52h not supported");

#ifdef WINDOWS
    /* get protected-mode pointer to DOS internal variable table */
    doslist = map_real(doslist, sizeof(ListOfLists));

    open_display(app);
#endif

    /* This block of code double-checks that everything is ok */
    /* NUL is part of DOSLIST, not a pointer, so don't need to map */
    if (_fmemcmp(doslist->nul.u.name, "NUL     ", 8) != 0)
        fail("NUL name wrong");
    if (! (doslist->nul.attr & IS_NUL))
        fail("NUL attr wrong");
```

Figure 8: Walking the device chain in protected-mode *Windows* is achieved by DEV.C.

DEV.C 2 of 2

```
#ifdef WINDOWS
    /* CON is pointer, so need to map */
    dd = map_real(doslist->con, sizeof(DeviceDriver));
#else
    dd = doslist->con;
#endif
    if (_fmemcmp(dd->u.name, "CON     ", 8) != 0)
        fail("CON name wrong");
    if (! (dd->attr & CHAR_DEV))
        fail("CON attr wrong");
#ifdef WINDOWS
    free_mapped_seg(dd);
#endif

#ifdef WINDOWS
    /* CLOCK$ is also pointer, so need to map */
    dd = map_real(doslist->clock, sizeof(DeviceDriver));
#else
    dd = doslist->clock;
#endif
    if (_fmemcmp(dd->u.name, "CLOCK$  ", 8) != 0)
        fail("CLOCK$ name wrong");
    if (! (dd->attr & IS_CLOCK))
        fail("CLOCK$ attr wrong");
#ifdef WINDOWS
    free_mapped_seg(dd);
#endif

    /* print out device chain */
    dd = &doslist->nul;
#ifdef WINDOWS
    for (;;)
    {
        printf("%Fp    ", DosProtToReal(dd)); /* print real-mode addr */
        if (dd->attr & CHAR_DEV)
            printf("%.8Fs\r\n", dd->u.name);
        else
            printf("Block dev: %u unit(s)\r\n", dd->u.blk_cnt);
        next = dd->next;            /* get next pointer */
        /* first time through, this will free selector to doslist */
        free_mapped_seg(dd);        /* THEN free rmode seg */
        if (FP_OFF(next) == -1)     /* is there a next? */
            break;
        dd = map_real(next, sizeof(DeviceDriver));  /* map it */
        Yield();    /* no message loop in this program, so yield */
    }
#else
    do {
        printf("%Fp\t", dd);
        if (dd->attr & CHAR_DEV)
            printf("%.8Fs\n", dd->u.name);
        else
            printf("Block dev: %u unit(s)\n", dd->u.blk_cnt);
        dd = dd->next;
    } while (FP_OFF(dd->next) != -1);
#endif

#ifdef WINDOWS
    if (mapped = get_mapped())
        printf("%u remaining mapped selectors!\r\n", mapped);
    show_display();
    return mapped;  /* 0 indicates success */
#else
    return 0;
#endif
}
```

DEV.C PARTIAL LISTING

```
ListOfLists far *doslist;
DeviceDriver far *dd;
// ...
dd = &doslist->nul;
for (;;)
{
    printf("%Fp    ", DosProtToReal(dd)); /* print real-mode addr */
    if (dd->attr & CHAR_DEV)
        printf("%.8Fs\r\n", dd->u.name);
    else
        printf("Block dev: %u unit(s)\r\n", dd->u.blk_cnt);
    next = dd->next;            /* get next pointer */
    /* first time through, this will free selector to doslist */
    free_mapped_seg(dd);        /* THEN free rmode seg */
    if (FP_OFF(next) == -1)     /* is there a next? */
        break;                  /* if not, we're done */
    dd = map_real(next, sizeof(DeviceDriver));  /* map it */
    Yield();  /* no message loop in this program, so yield */
}
```

Figure 9: The loop within DEV.C that walks the device chain. For an idea of the differences between real- and protected-mode, compare this to Figure 2.

later use before calling map_real().

Instead, however, DEV.C calls the DosProtToReal() function that was implemented in DPMI.C. Given a protected-mode pointer, this function attempts to return the corresponding real-mode pointer. The word "attempts" is important here, because if the base address for the protected-mode pointer is above the 1MB Maginot Line, then the protected-mode pointer has no real-mode counterpart.

In any case, DosProtToReal() attempts to make the conversion. Just for variety, rather than calling the dpmi_get_descriptor() function, it calls an undocumented *Windows* function, GetSelectorBase(). If the base address of the selector is greater than FFFFFh, then the pointer cannot be converted. Otherwise, DosProtToReal() combines the selector's base address with the protected-mode pointer's offset to form the desired real-mode pointer.

The GetSelectorBase() function used here is one of several such undocumented *Windows* functions. The function prototypes for several others are shown in DPMI.H (Figure 6). In fact, I could have used the SetSelectorBase() and SetSelectorLimit() functions, together with the documented *Windows* function AllocSelector(), and avoided DPMI entirely.

With regard to using undocumented functions, it's well to remember that there is no guarantee that they will continue to be supported in future releases of *Windows*—or even that they work 100 percent of the time in this release. That may be why they're undocumented in the first place!

On the other hand, these undocumented *Windows* functions really are far more convenient than the raw DPMI interrupt 31h functions. Further, in my experience, they have always worked—or at least worked as well as anything else in *Windows* 3.0. The tiny UNDOCSEL.C program listed in Figure 10 shows how these functions would be used to print out the BIOS timer-tick count at absolute memory location 46Ch (real-mode pointer 0040:006C).

Map_real() also makes use of an undocumented *Windows* feature: the __0000H selector. This is a hard-wired selector maintained by *Windows* that maps the 64K block of memory beginning at absolute memory location zero. Since many of the DOS device headers will be located in this first 64K of memory, it just made sense to use the *Windows* selector in this case.

Microsoft's *Windows Guide to Programming* (Chapter 16, "More Memory Management") describes global selectors such as __B000H and __B800H. These are used by full-screen *Windows* applications that do direct screen writes, such as Turbo Debugger for *Windows* (TDW) and the new single-screen version of CodeView for *Windows* (CVW). Note, however, that the Microsoft documentation does not mention the extremely useful __0000H selector.

UNDOCSEL.C **COMPLETE LISTING**


```
/*
UNDOCSEL.C -- illustrates undocumented Windows selector functions

cl -c -AS -G2sw -Oais -Zpe undocsel.c printf.c
link /align:16 undocsel printf,undocsel,,/nod slibcew libw,win.def
rc undocsel.exe

Copyright (c) 1991 Ziff Communications Co.
    PC Magazine * Andrew Schulman
*/

#include <windows.h>
#include <dos.h>
#include "printf.h"

/* undocumented Windows functions */
extern DWORD FAR PASCAL GetSelectorBase(unsigned sel);
extern DWORD FAR PASCAL GetSelectorLimit(unsigned sel);
extern void FAR PASCAL SetSelectorBase(unsigned sel, DWORD base);
extern void FAR PASCAL SetSelectorLimit(unsigned sel, DWORD limit);

#define FAIL(s) return MessageBox(NULL, s, "UNDOCSEL", MB_OK), 0

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpszCmdLine, int nCmdShow)
{
    unsigned sel, _ds;
    unsigned long far *pticks;
    if (! (GetWinFlags() & WF_PMODE))
        FAIL("This program requires Windows Standard or Enhanced mode");
    open_display("Undocumented Selector Function Test");
    _asm mov _ds, ds
    sel = AllocSelector(_ds);          // copy DS
    printf("sel=%04X\n", sel);
    SetSelectorBase(sel, 0x400);       // BIOS data area
    SetSelectorLimit(sel, 0xFFFF);     // only 64k allowed
    FP_SEG(pticks) = sel;
    FP_OFF(pticks) = 0x6c;             // BIOS timer tick
    printf("base=%08lx limit=%08lx ticks=%08lX\n",
        GetSelectorBase(sel), GetSelectorLimit(sel), *pticks);
    show_display();
    FreeSelector(sel);
}
```

Figure 10: The source code, UNDOCSEL.C, that uses an undocumented *Windows* function to print out the BIOS timer-tick count.

According to the Microsoft documentation, these global selectors are usable only from an assembly language program. The code in map_real() shows one way that __0000H can be accessed from a C program, however, and the other global selectors can be accessed in a similar fashion. One note of caution is in order: There is another undocumented selector, __0040H, which you would think mapped the BIOS data area at address 400h. In fact, this selector maps only the 2FFh bytes at address 0. Another undocumented selector, __ROMBIOS, is merely an alias for __F000H.

GlobalDOSAlloc() will be very important for *Windows* developers who need access to conventional memory.

WINDOWS AND DPMI

Having now seen how to call DPMI from *Windows* programs and how *Windows* provides its own undocumented layer on top of DPMI, it is natural to ask how DPMI fits into the *Windows* API.

You won't see DPMI mentioned in Microsoft's *Windows* documentation. In fact the Microsoft documentation barely refers to protected mode or to the fact that *Windows* in Standard and 386 Enhanced modes is a DOS extender. For many years, *Windows* essentially *faked* protected mode in software and even required developers to do a lot of work to help maintain the fiction. Now that *Windows* programs really do run in protected mode, most of the memory-management nonsense formerly required of *Windows* programs has disappeared. However, Microsoft's documentation barely mentions this either, and it downplays the massive change wrought by *Windows* 3.0 under the diplomatic title, "improved memory management."

Microsoft's only explicit mention of DPMI is a brief five-page document titled "Windows INT 21H and NetBIOS Support for DPMI," which is included in a packet of *Microsoft Windows* development notes (Part No. 050-030-313).

According to this rather obscure document, only a handful of DPMI functions should be used from *Windows* programs. The supported functions (which are listed on page 390 of *PC Magazine*'s February 26, 1991, issue) are those that set and get real-mode interrupt vectors, generate real-mode software interrupts, and call real-mode code.

For example, INT 31h AX=0300h (Simulate Real Mode Interrupt), together with the *Windows* function GlobalDosAlloc(), can be used for generating software interrupts not supported in protect mode. We will see an example of this toward the end of this article, in the short program TRUENAME.C.

According to Microsoft, apart from those listed, no other DPMI functions "are required for *Windows* applications, since the Kernel provides functions for allocating memory, manipulating descriptors, and locking memory."

Unfortunately, many developers have found that the *Windows* Kernel, even combined with the few "supported" DPMI functions, does *not* in fact provide everything they need to talk to real-mode TSRs, network drivers, and the like.

Again, the problem is this: Given a block of memory at an absolute real-mode address (such as the BIOS data area or the DOS device chain), how do you access this memory from your protected-mode *Windows* program? Even with the undocumented __0000H selector and the documented selectors such as __A000H, there is still an inaccessible hole between 10000h and A0000h!

The choice is either to use the undocumented SetSelectorBase() and SetSelectorLimit() functions, as in UNDOCSEL (Figure 10), or to use DPMI directly. The ability to map an arbitrary real-mode address into a program's protected-mode ad-

WINDOWS AND PRINTF()

by Andrew Schulman

The first problem you face when porting a PC program to *Microsoft Windows* is that you can't use such familiar output functions as printf() in C or Write() in Turbo Pascal. Instead, you must employ *Windows* functions like TextOut() or DrawText(). And, of course, you must completely restructure the program so that it responds to *Windows* "events."

Of course? The familiar claim that "it takes 100 lines even to say 'hello world' in *Windows*," is often repeated even by die-hard *Windows* evangelists. (The latter may indeed see such complexity as a virtue!) This gives the mistaken impression that even the smallest one-liner must be restructured as a full-blown *Windows* program, complete with a WndProc message handler and calls to the RegisterClass() and CreateWindow() functions.

Such a misconception makes it extremely difficult for newcomers to get started in *Windows* programming, and it inhibits experienced developers from trying to write tiny utilities or throwaway experiments.

It is true that if you use Microsoft C or Borland C++, you'll find that all the standard input/output (stdio) functions, such as printf(), puts(), and gets(), have been yanked from the *Windows* libraries. Trying to call one of these functions from a *Windows* program causes the linker to generate an "unresolved external."

But there is nothing magical about printf() or any other stdio function; stdio libraries can be built *on top of* the *Windows* API. For example, using Borland's *Turbo Pascal for Windows* (TPW), if you simply change the Pascal statement "Uses Crt" in a program to "Uses WinCrt," you've essentially created an instant *Windows* application: all your calls to the Pascal Write() statement go to a *Windows* window. Obviously, the same thing can be done in C or C++. In fact one wonders why Borland hasn't done it. In any case, it's relatively easy to write a module that allows its users to bring up *Windows* applications without rubbing their noses in the *Windows* API.

As we saw in last issue's Lab Notes, the *Windows* MessageBox() function can produce an entire screen of output. This makes it handy for tiny utilities like DEV. Alternatively, you can take advantage of *Windows*' multitasking and interprocess messaging to use another program (such as Notepad) as

PRINTF.H COMPLETE LISTING

```
/* PRINTF.H */

BOOL open_display(char *appname);
BOOL show_display(void);
int printf(const char *fmt, ...);
BOOL notepad(char far *s);
```

Figure A: The header file for the *Windows* version of printf().

PRINTF.C 1 of 2

```
/*
PRINTF.C -- simple output for small Windows programs,
using MessageBox() or WinExec()/SendMessage()

Copyright (c) 1991 Ziff Communications Co.
    PC Magazine * Andrew Schulman
*/

#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <windows.h>
#include "printf.h"

#define BUF_SIZE        1024

static char *str, *app;
static unsigned cap, len;
static int lines;

BOOL open_display(char *appname)
{
    app = appname;
    cap = 128;
    if (! (str = malloc(cap)))
        return FALSE;
    *str = lines = len = 0;
    return TRUE;
}

/* Maximum number of lines that MessageBox will hold */
static int max_lines(void)
{
    TEXTMETRIC tm;
    HWND hWnd = GetActiveWindow();
    HDC hDC = GetWindowDC(hWnd);
    if (hDC == NULL)
        return 0;
    GetTextMetrics(hDC, &tm);
    ReleaseDC(hWnd, hDC);
    return (GetSystemMetrics(SM_CYFULLSCREEN) /
        (tm.tmHeight + tm.tmExternalLeading)) - 5;
}

BOOL show_display(void)
{
    if (lines <= max_lines())
        MessageBox(NULL, str, app, MB_OK);
    else
        notepad(str);
    free(str);
    return TRUE;
}

static BOOL append(char *s2)
{
    char *s3;
    if (((len += strlen(s2)) < cap) && strcat(str, s2))
        return TRUE;
    cap = len + 128;
    if (! (s3 = malloc(cap)))
        return FALSE;
    strcpy(s3, str);
    strcat(s3, s2);
    free(str);
    str = s3;
    return TRUE;
}

int nlines(char *s2)
{
    int c, n = 0;
    while (c = *s2++)
        if (c == '\n')
            n++;
```

Figure B: PRINTF.C provides a *Windows* version of printf() for simple output.

CONTINUES

CONTINUED

PRINTF.C 2 of 2

```
    return n;
}

int printf(const char *fmt, ...)
{
    static char s2[BUF_SIZE];
    int len;
    va_list marker;
    va_start(marker, fmt);
    len = vsprintf(s2, fmt, marker);
    lines += nlines(s2);
    va_end(marker);
    append(s2);
    return len;
}

BOOL notepad(char far *s)
{
    HWND notepad;
    HWND edit_ctrl;
    if (WinExec("notepad.exe", SW_SHOWNORMAL) < 32)
        return FALSE;
    notepad = FindWindow(NULL, "Notepad - (untitled)");
    edit_ctrl = GetFocus();
    SendMessage(notepad, WM_SETTEXT, 0, (char far *) app);
    SendMessage(edit_ctrl, WM_SETTEXT, 0, (char far *) s);
    return TRUE;
}

#ifdef TESTING
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpszCmdLine, int nCmdShow)
{
    int i;
    open_display("GetSystemMetrics");
    for (i=0; i<37; i++)
    {
        printf("%d\t%d\r\n", i, GetSystemMetrics(i));
        Yield();
    }
    show_display();
}
#endif
```

your "display engine." Functions such as printf() can then be written on top of, and in terms of, such lower-level capabilities.

DEV.C, listed in Figure 8 of the main article, looks a lot simpler than most *Windows* source code. It doesn't register a class, call CreateWindow(), or handle messages. Instead, it calls printf(). When compiling for *Windows*, however, it uses the new version of printf() supplied by PRINTF.H and PRINTF.C, listed in Figures A and B. Here, printf() simply accumulates text until a call to show_display(). The show_display() function uses the *Windows* MessageBox() function.

If there is more text than MessageBox() can handle, then PRINTF.C uses WinExec() to fire up a copy of Notepad and sends the text to Notepad using the WM_SETTEXT message and the SendMessage() function. The ability of one program to send text to another shows, incidentally, that *Windows* is genuinely message-based: "messages" are a true form of interprocess communication, not just an elaborate way of describing a function call.

Windows itself can sometimes help in the task of providing a *Windows*-based printf(). The *Windows* dynamic-link library USER.EXE provides the functions wsprintf() and wvsprintf(). Like the vsprintf() of ANSI C, the wvsprintf() function can be used to construct other functions—such as printf()—that take a variable number of arguments. Because wvsprintf() resides in a DLL, using this *Windows* version of the function can reduce the size of your executable.

Unfortunately, wvsprintf() does not support several useful "masks," such as %Fp. It is also sometimes awkward to use: Since it resides in a DLL, it expects all strings to be passed as far pointers. In PRINTF.C, therefore, I ended up using the non-*Windows* vsprintf() function.

In porting DEV.C to *Windows*, in any case, you can simply assume that you have a function called printf() that takes the same parameters and returns the same value (almost always ignored!) as the "regular" printf(), but that somehow "does the right thing" in displaying output under *Windows*. Thus, it really is possible to write simple *Windows* programs! ■

dress space was left out of the published API, making DPMI even more necessary for *Windows* developers than Microsoft's brief description would indicate.

USING GLOBALDOSALLOC()

One documented *Windows* function that will be quite important for many developers who need to access conventional memory or call real-mode TSRs and drivers is GlobalDosAlloc(). This function allocates a block of conventional memory, returning both a real-mode paragraph address and a protected-mode selector to the memory.

Suppose, for example, that you're writing a piece of network software for *Windows* using a specification like NetBIOS or Novell IPX/SPX. In addition to needing dpmi_rmode_intr() (or its underlying DPMI Simulate Real Mode Interrupt call, INT 31h AX=0300h) to generate the real-mode software interrupt, you must also pay attention to any buffers expected by the network driver.

The NetBIOS INT 5Ch interface, for instance, expects a pointer to a Network Control Block (NCB) in ES:BX. When calling this from protected mode, even though you are going through the DPMI Simulate Real Mode Interrupt call, you must ensure (1) that the NCB is in conventional memory and (2) that ES:BX has a real-mode pointer. On the other hand, your program must manipulate the same NCB using a protected-mode pointer.

That's where GlobalDosAlloc() comes in. By allocating out of conventional memory and giving back both a real-mode paragraph address and a protected-mode selector, it's exactly the function you need to use in conjunction with dpmi_rmode_intr(). I found the *way* that GlobalDosAlloc() returns these two different values annoying, so I constructed another function on top of it, DosAllocRealSeg(). (Like the earlier DosMapRealSeg(), the name comes from the Phar Lap API.)

The GlobalDosAlloc() function (implemented via the more convenient DosAllocRealSeg()) and dpmi_rmode_intr(), are used in TRUENAME.C (Figure 11), the last tiny *Windows* program presented here. Because many readers don't ha... NetBIOS, I have again used an undo... mented DOS call—the one category I know is available on every DOS machine, yet which is not supported directly from

TRUENAME.C **COMPLETE LISTING**


```
/* truename.c */

#include <windows.h>
#include <stdlib.h>
#include <ctype.h>
#include "printf.h"
#include "dpmi.h"

#define fail(s)          { MessageBox(NULL, s, "TRUENAME", MB_OK); exit(1); }

#define MAKEP(seg, ofs) ((void far *) MAKELONG((ofs), (seg)))

char far *truename(char far *s, char far *d)
{
            RMODE_CALL r;
            unsigned para, sel;
    char far *s2;

    /* INT 21h AH=60h doesn't like leading or trailing blanks */
    while (isspace(*s)) s++;
    s2 = s;
    while (*s2) s2++;
    s2--;
    while (isspace(*s2)) *s2-- = 0;

            if (DosAllocRealSeg(256, &para, &sel) != 0)
                        fail("Couldn't allocate real-mode memory");
            memset(&r, 0, sizeof(r));
            r.eax = 0x6000;
            r.ds = r.es = para;
            r.esi = 0;
            r.edi = 128;
            lstrcpy(MAKEP(sel, 0), s);
            if (! dpmi_rmode_intr(0x21, 0, 0, &r))
                        fail("DPMI real-mode intr failed");
            lstrcpy(d, (r.flags & 1) ? "<Invalid>" : (char far *) MAKEP(sel, 128));
            if (DosFreeRealSeg(sel) != 0)
                        fail("Couldn't free real-mode memory");
            return d;
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
            LPSTR lpszCmdLine, int nCmdShow)

            char buf[128];
            if (! (GetWinFlags() & WF_PMODE))
                        fail("This program requires Windows Standard or Enhanced mode");
            if (! (lpszCmdLine && *lpszCmdLine))
                        fail("syntax: truename <pathname>");
            sprintf(buf, "TRUENAME %Fs", lpszCmdLine);
            open_display(buf);
            printf("%Fs\n", truename(lpszCmdLine, buf));
            show_display();
```

Figure 11: TRUENAME.C shows how to pass a conventional memory buffer to a real-mode interrupt.

protected-mode *Windows*.

The TRUENAME program takes a pathname from the command line (for example, the Program Manager Run . . . command) and uses INT 21h AH=60h to return the "true" underlying pathname, with any DOS ASSIGNs, SUBSTs, JOINs, or network mappings resolved. On my machine at work, for example, drive F: is actually located on a Sun SPARCstation, and running TRUENAME F: displays something like

```
\\EXPORT\DOS\A_ANDREW.
```

The reason this program is a good illustration of using GlobalDosAlloc() together with dpmi_rmode_intr(), is that INT 21h AH=60h expects a source pathname in DS:SI (such as "F:."), and a pointer to a 128-byte destination buffer in ES:DI (into which it will place a string such as "\\EXPORT\DOS\A_ANDREW"). Before calling INT 21h AH=60h, the program must allocate a conventional-memory transfer buffer, copy the source string into it, put its real-mode address in DS:SI, and put the address of another conventional-memory transfer buffer (the destination) in ES:DI. When dpmi_rmode_intr() returns after INT 21h AH=60h has been invoked, you can copy the results out of the conventional-memory buffer. Of course, you must do this copying using the protected-mode pointer to the buffer.

A GOOD PROBLEM

There are many ways that *Windows* programs can use DPMI. Other DPMI services I haven't discussed here let programs hook real-mode interrupt vectors, enhance debugging with access to protected-mode data structures such as the Global Descriptor Table (GDT), and catch GP faults and other exceptions.

This last-mentioned possibility is really quite interesting. I earlier talked about the GP fault and UAE as if they were an immutable fact of protected-mode life. In fact, however, the GP fault is nothing but an interrupt 0Dh. DPMI makes a distinction between interrupts and processor exceptions and provides functions for handling exceptions like the GP fault.

The DPMI Set Processor Exception Handler Vector (INT 31h AX=0203h) and the Get Processor Exception Handler Vector (INT 31h AX=0202h) calls can be used to replace *Windows*' annoying UAE with your own GP fault handler. Such a handler might display a message box politely asking the end user of a program that has GP faulted to call tech support, for example. Anything would be better than hitting the user with the "Unrecoverable Application Error" message!

Amidst all the complexities of protected-mode *Windows*, it's a good idea to conclude by remembering the tremendous advantages brought about by the demolition of the 1MB "Berlin Wall." These far outweigh any disadvantages resulting from the sudden inability to examine address 400h with the pointer 0000:0400, for example. In any case, this discussion has shown that you can still examine the memory at 400h; it's just that you can't do it with the pointer 0000:0400 or 0040:0000. Instead, you have to use whatever equivalent protected-mode pointer is returned from map_real(), from the underlying DPMI calls, or from a combination of undocumented *Windows* calls.

The problem with low-level access from protected mode reflects PC programming's move to a new, higher level. We have broken the 640K barrier, but we have not shed our massive investment in DOS software. The programming problems demonstrated in this and the previous Lab Notes represent the *successful* compromise between protected mode and DOS. More and more programs are running in protected mode, yet the industry's investment in DOS has been preserved. This is a good problem to have! ■

Andrew Schulman is an engineer and writer at Phar Lap Software, in Cambridge, Massachusetts. Editor of the recent book Undocumented DOS, *he is also a contributor to* Extending DOS, *edited by Ray Duncan.*